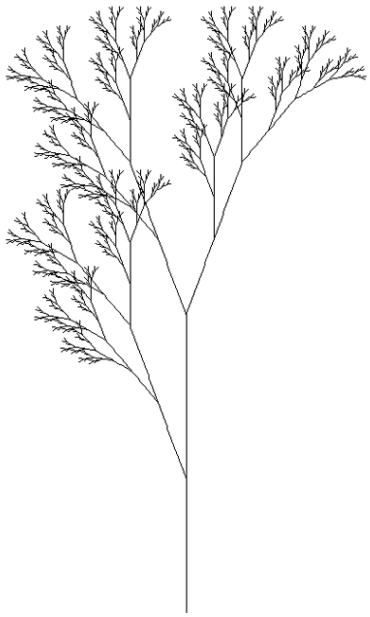
TP INFO 4 | Initiation à la programmation récursive

On expose les fondements de la programmation récursive, ses avantages et ses inconvénients en les illustrant au moyen des algorithmes classiques.



4 Initiation à la programmation récursive 1
1 Introduction 2
2 Fonctions récursives 2
2.1 Exemples classiques 3
2.2 Déroulemement d'un appel récursif 4
3 Exercices 6
4 Indications 11

1. Introduction

Le mot *récursivité* vient de la racine latine *currere*, « courir ». L'idée est de *revenir en arrière*, on la retrouve également dans le vocabulaire judiciaire (*recours*). En première approche, on peut considérer la récursivité comme une version informatique de la récurrence.

Par exemple, pour calculer $n!$ en n'utilisant que la multiplication, on peut utiliser :

- *Un traitement itératif* :

$$\text{on calcule } 1,\ 1\times 2 \text{ puis } (1\times 2)\times 3,\ \text{etc. jusqu'à } (1\times 2\times\cdots\times(n-1))\times n$$

- *La relation de récurrence* $n! = n\times(n-1)!$ *et son initialisation* $0! = 1$ *:*

$$\begin{cases} \text{si } n\neq 0, \text{ alors on calcule } (n-1)! \text{ et on multiplie le résultat par } n.\\ \text{Si } n = 0, \text{ alors on renvoie } 1. \end{cases}$$

La plupart des langages de programmation offrent la possibilité de coder naturellement cette récurrence

Cela donne les programmes suivants en Python :

```
def factIt(n):
    p=1
    for i in range(2,n+1):
        p=p*i
    return p
```

```
def factRec(n):
    if n==0:
        return 1
    else:
        return n*factRec(n-1)
```

La fonction `factRec` est dite *récursive* et s'utilise, comme toute autre fonction, au moyen d'un appel :

```
>>> factIt(5)
120
>>> factIt(1)
1
```

```
>>> factIt(0)
1
>>> factRec(0)
1
```

```
>>> factRec(1)
1
>>> factRec(5)
120
```

La notion actuelle de fonction récursive a ses origines dans les travaux sur la calculabilité de Skolem, Gödel et Kleene dans les années 1920-1940.

2. Fonctions récursives

Nous n'entrerons dans aucun développement de logique formelle concernant la calculabilité et nous contenterons de la « définition » suivante :

Définition 4.0. Fonction récursive

Une fonction récursive est une fonction faisant appel à elle-même.

Afin que la série des appels récursifs ne soit pas infinie, il convient d'identifier un ou plusieurs cas pour lesquels aucun appel récursif ne sera effectué. Dans le cas de la fonction `factRec`, il s'agit du cas où $n = 0$ qui correspond à une initialisation. On les appelle *cas de base* de la fonction récursive.

2.1. Exemples classiques

Très naturelle dans certains contextes (calcul des termes d'une suite récurrente par exemple), la récursivité est parfois plus subtile à mettre en place.

- *Récurrences d'ordre un.*

  Considérons par exemple la suite définie par $u_0 = 1$ et $u_{n+1} = \sin u_n$ pour tout $n \in \mathbb{N}$.

```
def u(n):
    if n==0:
        return 1
    else:
        return sin(u(n-1))
```

Ici, le cas de base est simple (il s'agit de la condition initiale) et l'on effectue un unique appel récursif.

```
>>> u(10)
0.46295789853781183
```

  Il est possible d'effectuer plus d'un appel récursif dans la fonction.

- *Récurrences d'ordre deux et plus.*

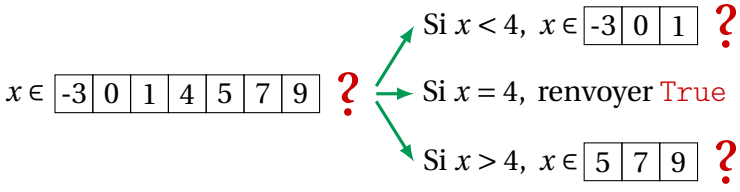
  Prenons l'inusable exemple de la suite de Fibonacci $(f_n)_{n\geqslant 0}$.

```
def fib(n):
    if n<=1:
        return n
    else:
        return fib(n-1)+fib(n-2)
```

Les cas de base sont évidents (les conditions initiales) et l'on effectue deux appels récursifs.

```
>>> fib(10)
55
```

- Nous avons déjà vu en TP un algorithme naturellement récursif : la recherche dichotomique dans une liste triée. Pour déterminer si un nombre x appartient à une liste triée numérique t dans l'ordre croissant, on la coupe en deux en son milieu $t[m]$ et, selon la position relative de $t[m]$ et x, on renvoie `True` ou bien l'on recommence avec une des deux « moitiés » de t.



On utilise le *slicing* afin de « construire » les portions de liste de l'algorithme dichotomique. Le cas de base peut être celui où `t` est vide ou bien réduit à un élément (valable si on n'applique pas la fonction à une liste vide).

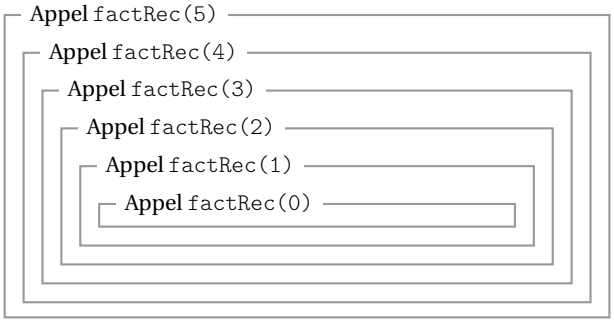
```
def biSearchRec(t,x):
    if len(t)==0:
        return False
    else:
        m=(len(t)-1)//2
        if x==t[m]:
            return True
        elif x>t[m]:
            return biSearchRec(t[m+1:],x)
        else:
            return biSearchRec(t[:m],x)
```

2.2. Déroulement d'un appel récursif

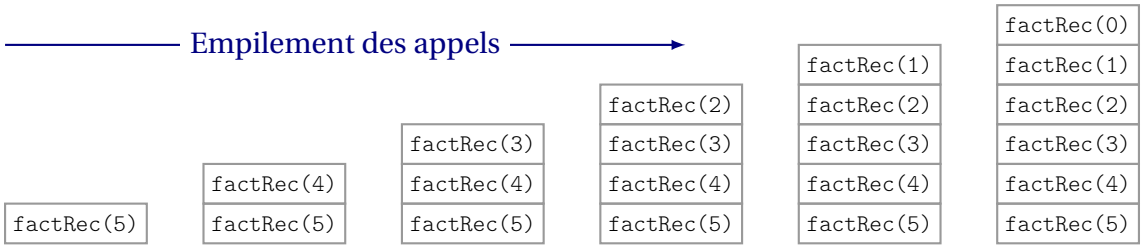
Quant on appelle une fonction récursive, cette dernière est en fait appelée à plusieurs reprises. Pour gérer un programme où plusieurs fonctions sont appelées, l'interpréteur utilise *une pile d'exécution*. Il y emmagasine des valeurs et des adresses permettant de garder la trace de l'endroit où chaque fonction active[1] doit retourner à la fin de son exécution.

Reprenons l'exemple de la fonction `factRec` de l'introduction (cf. 2). Considérons l'appel `factRec(5)`. Avant de renvoyer quoi que ce soit, il engendre un appel `factRec(4)` qui lui-même engendre un nouvel appel `factRec(3)` et ainsi de suite jusqu'à l'appel de `factRec(0)`.

Les appels sont imbriqués les uns dans les autres comme des poupées russes.



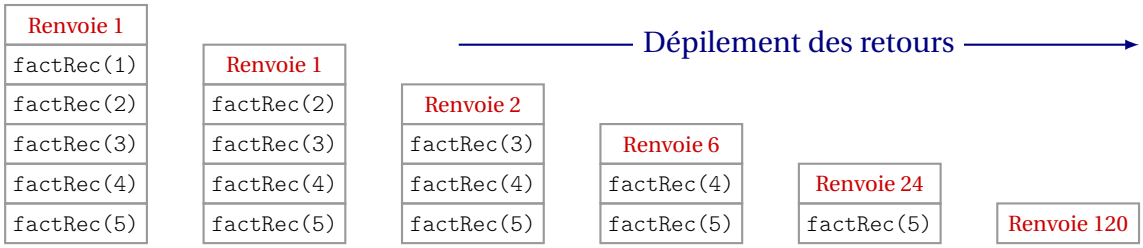
Visualisons cela au moyen de la pile d'exécution :



L'appel `factRec(0)` n'engendre aucun nouvel appel et se solde par le renvoi de 1 dans l'appel `factRec(1)`, qui renvoie 1×1 dans l'appel `factRec(2)`, qui renvoie $2 \times 1 = 2$ dans l'appel `factRec(3)`,

1. Les fonctions actives sont celles qui ont été appelées, mais n'ont pas encore terminé leur exécution.

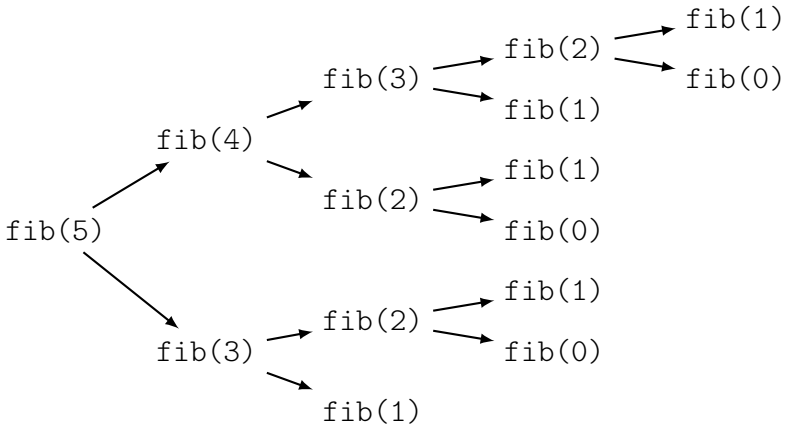
qui renvoie $3 \times 2 = 6$ dans l'appel `factRec(4)`, qui renvoie $4 \times 6 = 24$ dans l'appel `factRec(5)` qui renvoie $5 \times 24 = 120$.



La taille de la pile est bien-sûr limitée par la mémoire finie de l'ordinateur. En Python, la taille de la pile est limitée par défaut à 1000, ce qui est vite atteint en cas d'appels récursifs multiples dans la définition de la fonction. Il est cependant possible d'augmenter sa taille pour les besoins d'une exécution[2].

Dans le cas où la fonction compte plus d'un appel récursif, il est intéressant de représenter les appels au moyen d'un arbre.

Revenons à la fonction `fib` calculant les termes de la suite de fibonacci (cf. page 3). L'appel `fib(5)` peut se résumer par l'arbre ci-contre. On y voit en particulier *l'inflation du nombre d'appels*. Par exemple, `fib(2)` est calculé trois fois.



Il faut donc éviter d'utiliser la fonction récursive `fib`, donnée dans le paragraphe d'exemples, pour calculer les termes de la suite de Fibonacci : pour valeurs petites de n, le temps de calcul sera bien trop long.

On retiendra les deux conseils suivants :

Écueils de la récursivité et recommandations

- On évitera des appels récursifs multiples sur les mêmes variables (cf. l'exempe de Fibonacci).
- On identifiera clairement les cas de base de la fonction récursive : il faut s'assurer que tout appel récursif aboutira à l'un d'entre eux.

2. En utilisant la fonction `setrecursionlimit` du module **sys**

3. Exercices

1 — *Suite de Prouhet-Thue-Morse*

On définit la suite (t_n) par $t_0 = 0$, $\forall n \in \mathbb{N}$, $\begin{cases} t_{2n} &= t_n \\ t_{2n+1} &= 1 - t_n \end{cases}$

Écrire une fonction récursive `ptm(n)` renvoyant t_n pour tout $n \in \mathbb{N}$.

2 — *Algorithmes d'exponentiation*

La seconde méthode est connue sous le nom *d'exponentiation rapide*.

1. Écrire une fonction récursive `expo(a,n)` renvoyant a^n pour $a \in \mathbb{Z}$ et $n \in \mathbb{N}$.
2. Pour un réel positif a et un entier n, on remarque que $a^n = \begin{cases} \left(a^{\frac{n}{2}}\right)^2 & \text{si } n \text{ est pair} \\ a\left(a^{\frac{n-1}{2}}\right)^2 & \text{si } n \text{ est impair} \end{cases}$

   En déduire une nouvelle fonction récursive `quickExpo(a,n)` renvoyant a^n pour $a \in \mathbb{Z}$ et $n \in \mathbb{N}$.

3 — *Retour à Fibonacci : amélioration par vectorialisation* f

On considère la suite de Fibonacci :

$$f_0 = 0,\ f_1 = 1,\ \forall n \in \mathbb{N},\ f_{n+2} = f_{n+1} + f_n$$

Nous avons vu dans le cours qu'une fonction avec deux appels récursifs souffrait d'un temps de calcul trop long par rapport à une version itérative. Nous allons voir comment remédier à cet écueil en revenant à un seul appel récursif.

1. On pose $X_n = (f_n, f_{n+1})$ pour tout $n \in \mathbb{N}$. Déterminer une fonction $\phi : \mathbb{R}^2 \to \mathbb{R}^2$ telle que
$$\forall n \in \mathbb{N},\ X_{n+1} = \phi(X_n)$$
2. En déduire une fonction récursive `X` prenant en argument un entier n et renvoyant X_n sous la forme d'une liste.

4 — *Coefficients binomiaux* f

On étudie différents algorithmes récursifs de calcul des coefficient $\binom{n}{k}$. On renverra un résultat du type `int`.

1. Écrire une fonction récursive `binome1(n,k)` exploitant la relation de Pascal.
2. En remarquant que $\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$, écrire une fonction récursive `binome2(n,k)` plus efficace.

5 — *Calcul récursif du maximum d'une liste* *f*

On propose deux approches récursives pour évaluer le maximum d'une liste d'entiers.

1. En utilisant la définition par récurrence du maximum

$$\max(t_0,\dots,t_n) := \max\big(t_0, \max(t_1,\dots,t_n)\big)$$

   écrire une fonction récursive `maxRec` prenant en argument une liste d'entiers `t` et renvoyant son maximum. On se souviendra que `t[1:]` est la liste obtenue en supprimant le premier terme de `t`.

2. Écrire une fonction récursive `maxRecBis` prenant en argument une liste d'entiers `t` et un indice `i` renvoyant le maximum de `t[i:]`. On impose cette fois-ci que la récursion se fasse uniquement suivant la variable d'indice.

6 — *Algorithme de Hörner* *ff*

Soit $P = p_0 + p_1X + \cdots + p_nX^n$ un polynôme à coefficients réels et $a \in \mathbb{R}$. Le polynôme P sera représenté par la liste $(p_0,\dots,p_n)$ de ses coefficients. Pour calculer efficacement $P(a)$, Hörner a proposé la méthode suivante :

$$P(a) = p_0 + a\cdot\big(p_1 + a\cdot\big(p_2 + a\cdot\big(p_3 + a\cdot(\cdots)\big)\big)\big)$$

Par exemple, pour calculer $P(2)$ lorsque $P = p_3X^3 + p_2X^2 + p_1X + p_0$, on calcule successivement :

$$p_3,\ \ p_2+2p_3,\ \ p_1+2(p_2+2p_3) = p_1+2p_2+2^2p_3,\ \ p_0+2\big(p_1+2p_2+2^2p_3\big) = p_0+p_12+p_22^2+p_32^3 = P(2)$$

1. Écrire une fonction récursive `horner(t,a)` prenant en argument la liste `t` représentant le polynôme P et le réel a et renvoyant $P(a)$ selon la méthode de Hörner.
2. En quoi l'algorithme de Hörner est-il plus efficace que celui utilisé ci-dessous ?

```
def eval(P,a):
    s=0
    for i in range(len(P)):
        s=s+p[i]*a**i
    return s
```

7 — *Un algorithme glouton* *ff*

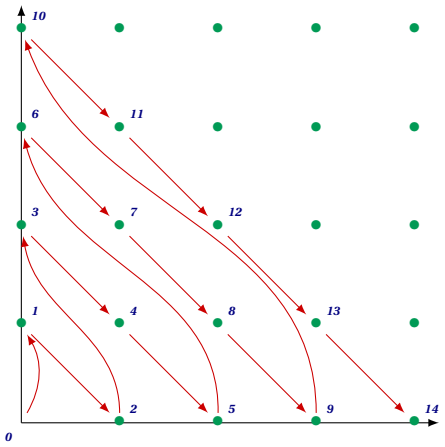
On revient au TP 3 et à l'algorithme glouton de paiement d'une somme n avec de devises de valeurs $v_0 > v_2 > \cdots > v_{p-1} = 1$. On code les données du problème au moyen de deux variables : `n` du type `int` (contenant la valeur de n) et `val` du type `list` (contenant dans l'ordre v_0, …, v_{p-1}). On renverra la solution sous la forme d'une liste `paiement` contenant $(x_0,\dots,x_{p-1})$ (correspondant à un paiement avec x_i fois la pièce de valeur v_i pour tout $i \in [\![0, p-1]\!]$).

Écrire une fonction récursive `monnaie(n,val)` renvoyant la liste `paiement` selon cette méthode.

8 — *Une énumération de $\mathbb{N}^2$* ***ff***

Il s'agit de coder la *célèbre* bijection de $\mathbb{N}$ sur $\mathbb{N}^2$ ainsi que sa réciproque.

On se place dans un plan muni d'un repère orthonormé, et on numérote chaque point de $\mathbb{N}^2$ par le procédé décrit ci-contre.



1. Écrire une fonction récursive `enumPtoN(x,y)` d'arguments x et y ayant pour résultat le numéro du point de coordonnées (x, y).
2. Écrire une fonction récursive `enumNtoP(n)` d'argument n et ayant pour résultat les coordonnées (x, y) du point numéroté par n.

REMARQUE : il n'est pas difficile d'expliciter cette bijection ϕ et son inverse ϕ^{-1}.

9 — *Tours de Hanoï* ***ff***

Sur une tablette sont disposées 3 tiges 1, 2, 3.

Sur la tige 1, on a empilé n disques de taille strictement décroissante. On veut déplacer cette pyramide vers la tige 3 en utilisant la règle suivante : on peut enlever un disque du sommet pour le mettre sur une autre tige à condition qu'il repose sur un disque de rayon plus grand ou sur aucun disque.

1. Écrire une fonction récursive `hanoi(n,o,i,d)` d'arguments `n`, `o` (tige d'origine), `i` (tige intermédiaire) et `d` (tige de destination) affichant les mouvements à effectuer (sous la forme d'une succession de lignes de la forme « De o en i ») pour effectuer le transfert des `n` disques de la tige `o` à la tour `d` en utilisant la tige `i` comme intermédiaire. L'appel `hanoi(n,1,2,3)` répondra alors à la question.
2. Dénombrer les mouvements renvoyés par l'appel `hanoi(n,1,2,3)`.

10 — *Énumération des permutations de $[\![0, n-1]\!]$* ***ff***

Une permutation f de $[\![0, n-1]\!]$ est par définition une bijection de $[\![0, n-1]\!]$ sur lui-même et représentée par la liste `[f(0),...,f(n-1)]` de ses images. Écrire une fonction récursive `enPerm(n)` renvoyant la liste des permutations de $[\![0, n-1]\!]$.

11 — *Un exemple de Backtracking* ***fff***

On se propose de déterminer tous les carrés magiques de taille 3 à coefficients dans $[\![1,9]\!]$, i.e. toutes les matrices à trois lignes et trois colonnes à coefficients dans $[\![1,9]\!]$ dont les sommes des lignes, des colonnes et des deux diagonales principales sont égales :

6	1	8
7	5	3
2	9	4

On codera systématiquement un tableau par une liste de listes. Pour l'exemple ci-contre :

```
[[6,1,8],[7,5,3],[2,9,4]]
```

On pourra utiliser la fonction `sum` qui prend en argument une liste d'entiers et en renvoie la somme.

1. Écrire une fonction `estMagique` prenant en argument une matrice carrée de taille trois et renvoyant `True` si elle est magique et `False` sinon.
2. On numérote les coefficients d'une matrice à trois lignes et trois colonnes de 0 à 8 en la parcourant de la première à la dernière ligne, de gauche à droite. Soit $n \in [\![0,8]\!]$. Sur quelle ligne et quelle colonne le coefficient numéroté n est-il situé ?

On va générer toutes les solutions en utilisant une variable `sol_p` pour contenir des solutions partielles au cours de l'algorithme. Une solution partielle est une matrice dont certains coefficients n'ont pas encore été choisis et sont laissés égaux à 0 par convention. Par exemple :

```
[[6,1,8],[7,5,0],[0,0,0]]
```

3. Écrire une fonction récursive `completer` prenant en arguments un entier `n`, une solution partielle `sol_p` et une liste de solutions `liste` qui ajoute à `liste` toutes les solutions dont les coefficients numérotés de 0 à $n-1$ sont identiques à ceux de `sol_p`.
4. Écrire un script permettant d'obtenir toutes les solutions. Combien en dénombre-t-on ?

12 — *Mots de Dyck* ***fff***

Soit $n \in \mathbb{N}^*$. On dit qu'une liste `t` est *un mot de Dyck de longueur* $2n$ si :

1. elle est de longueur $2n$;
2. ne comporte que des -1 et des 1 ;
3. comporte autant de 1 que de -1 ;
4. pour tout $k \in \{0,\dots,2n-1\}$, il y a dans `[t[0],...,t[k]]` au moins autant de 1 que de -1.

Par convention, la liste vide est le seul mot de Dyck de longueur nulle.

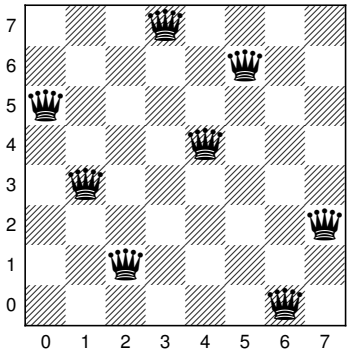
Il s'agit de toutes les formules de parenthésage correct. Par exemple, l'expression e_1 = `((()(()))` *correspond au mot de Dyck* `[1,1,-1,1,1,-1,1,-1,-1,-1]`. *De même, l'expression* e_2 = `()()` *correspond à* `[1,-1,1,-1]`.

On admet que tout mot de Dyck non vide `t` s'écrit de façon unique sous la forme `t=[1]+u+[-1]+v` où `u` et `v` sont deux mots de Dyck (éventuellement vides). Écrire une fonction récursive `enumDyck(n)` qui renvoie la liste de tous les mots de Dyck de longueur $2n$.

13 — *Le problème des huits reines* ***fff***

Le premier à formuler ce problème fut vraisemblablement le mathématicien allemand Johann Carl Friedrich Gauss (1777-1855).

Il s'agit de placer huit reines d'un jeu d'échecs sur un échiquier de 8 × 8 cases sans que les dames ne se menacent mutuellement. Les règles sont celles du jeu d'échecs et la couleur des pièces est ignorée. Par conséquent, deux reines ne devraient jamais partager la même rangée, colonne, ou diagonale. On peut trouver les solutions en explorant récursivement l'espace des configurations possibles. Le but de cet exercice est de dénombrer les solutions du problème des 8 reines. On numérote les lignes et les colonnes de 0 à 7. Une solution peut être codée au moyen d'une liste `pos` (variable globale). Pour tout `i`, la reine située sur la ligne `i` se trouve sur la colonne `j = pos[i]`.



Les tableaux `pos` seront remplis récursivement. Un `compteur` (variable globale) sera utilisé pour déterminer le nombre de solutions.

1. Écrire une fonction `conflit(i1,j1,i2,j2)` renvoyant sous la forme d'un booléen si les reines situées en `(i1,j1)` et `(i2,j2)` (numéro de ligne puis de colonne) sont en conflit.
2. Écrire une fonction `compatible(i,j)` renvoyant sous la forme d'un booléen si une reine en position `(i,j)` est compatible avec les reines déjà placées dans les lignes `0`, …, `i-1` (sauvegardées dans les `i` premières cases du tableau `pos`). La fonction doit renvoyer `True` quelque soit `j` lorsque `i=0`.
3. Écrire une fonction récursive `reine(i)` telle que l'appel `reines(0)` produise l'affichage des solutions (ie les tableaux `pos` correspondants aux solutions) suivi du nombre de solutions.
4. Déterminer le nombre de solutions au problème des huit reines.

4. Indications

1

L'entier n est le quotient dans les divisions euclidiennes de $2n$ et $2n+1$ par 2.

2

Tester la parité de n au moyen du reste dans la division euclidienne.

3

La fonction $\phi : (x, y) \mapsto (y, y + x)$ convient.

4

Au 2., on utilisera le quotient dans la division afin de renvoyer un résultat du type `int`.

5

Au 2., pour renvoyer le maximum de `t[i:]`, il suffit de calculer récursivement le maximum de `t[i+1:]` et de le comparer à `t[i]`.

6

Remarquer que $\mathrm{P}(a) = p_0 + a \times \mathrm{P}_1(a)$ où P_1 est le polynôme de coefficients $(p_1, \ldots, p_n)$.

7

Effectuer le premier paiement avec v_0 puis les autres récursivement.

8

Pour trouver le numéro du point (x, y), il suffit d'ajouter un à celui du point *précédent*.

9

Posez-vous la question suivante : si on sait résoudre le problème pour $n-1$ disques, comment peut-on le résoudre pour n ?

10

Pour générer toutes les permutations de 0, …, $n-1$, il suffit de décrire toutes les permutations de 0, …, $n-2$ et, pour chacune d'entre elles, insérer $n-1$ à toutes positions possibles (n au total).

11

Au 2., remarquer que $n = 3 \times i + j$ où i et j sont respectivement les indices de ligne et de colonne du coefficient numéroté n.

12

Il faut faire varier (u, v) dans l'ensemble des couples de mots de Dyck dont la somme des longueurs vaut $2n$ afin de générer tous les mots de Dyck de longueur $2n+2$.

13

Il s'agit d'un algorithme de Backtracking (cf. l'exercice sur les carrés magiques de taille trois). Le cas de base de `reine` est celui où $i = 8$, pour lequel on obtient une nouvelle solution.